UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

INSTITUTO DE INFORMATICA

Aula 31/1/23 - Sistemas Digitais

Prof. Fernanda Kastensmit

TRABALHO FINAL AHMES

Pontuação: 10 pontos (vale 1/4 da nota do semestre)

Nome: Giordano Souza de Paula Cartão UFRGS: 00308054

**Objetivo:** projetar e descrever em VHDL o processador Ahmes, implementar 2 programas em sua memória e mostrar através de simulação lógica sem e com atraso o funcionamento.

**PASSO 1:** 3 pontos

Descrever o DATAPATH do processador AHMES em VHDL em uma entidade apenas chamada de datapath\_ahmes.

***Cole aqui o código completo em VHDL do datapath***

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_unsigned.all;

use ieee.numeric\_std.all;

entity datapath is

port(

CLOCK: in std\_logic;

RESET: in std\_logic;

-- register control inputs

inc\_PC: in std\_logic;

load\_PC: in std\_logic;

load\_AC: in std\_logic;

load\_MA: in std\_logic; -- memory address register

load\_MD: in std\_logic; -- memory data register

load\_I: in std\_logic; -- instruction register

load\_N: in std\_logic;

load\_Z: in std\_logic;

load\_V: in std\_logic;

load\_C: in std\_logic;

load\_B: in std\_logic;

-- selectors

sel\_MUX\_MAR: in std\_logic\_vector(0 downto 0); -- selects which data reg\_MA should recieve (between reg\_PC or reg\_MD)

sel\_MUX\_MDR: in std\_logic\_vector(0 downto 0); -- selects which data reg\_MD should recieve (between reg\_MEM or reg\_AC)

sel\_ALU: in std\_logic\_vector(3 downto 0); -- selects which arithmetic operation ALU should execute

-- memory control

mem\_read: in std\_logic\_vector(0 downto 0);

mem\_write: in std\_logic\_vector(0 downto 0);

mem\_out: out std\_logic\_vector(7 downto 0);

-- ALU flags

reg\_N: out std\_logic;

reg\_Z: out std\_logic;

reg\_V: out std\_logic:= '0';

reg\_C: out std\_logic;

reg\_B: out std\_logic:= '0';

-- instruction flags (from reg\_I)

instruction\_flags: out std\_logic\_vector(23 downto 0));

end datapath;

architecture Behavioral of datapath is

-- registersf

signal reg\_PC: std\_logic\_vector(7 downto 0):= "00000000";

signal reg\_AC: std\_logic\_vector (7 downto 0):= "00000000";

signal reg\_MA: std\_logic\_vector(7 downto 0):= "00000000";

signal reg\_MD: std\_logic\_vector(7 downto 0):= "00000000";

signal reg\_I: std\_logic\_vector(7 downto 0):= "00000000";

signal MAR\_MUX\_out: std\_logic\_vector(7 downto 0):= "00000000";

signal MDR\_MUX\_out: std\_logic\_vector(7 downto 0):= "00000000";

signal ALU\_out: std\_logic\_vector(7 downto 0):= "00000000";

signal omem\_out: std\_logic\_vector(7 downto 0):= "00000000";

signal IR\_DECOD\_out: std\_logic\_vector(23 downto 0):= "000000000000000000000000";

-- AC flags (for upkeeping internal values until the register update on rising edge)

signal flag\_N: std\_logic:= '0';

signal flag\_Z: std\_logic:= '0';

signal flag\_V: std\_logic:= '0';

signal flag\_C: std\_logic:= '0';

signal flag\_B: std\_logic:= '0';

-- ALU

signal ALU\_X: std\_logic\_vector(7 downto 0):= "00000000"; -- recieves signals from reg\_AC

signal ALU\_Y: std\_logic\_vector(7 downto 0):= "00000000"; -- recieves signals from reg\_RDM;

signal ALU\_op: std\_logic\_vector(8 downto 0):= "000000000"; -- ALU operational signal

-- the extra bit is used for the overflow(V) flag

-- ALU operation constants

constant ALUNOP: std\_logic\_vector(3 downto 0):= "0000";

constant ALUADD: std\_logic\_vector(3 downto 0):= "0001";

constant ALUOR: std\_logic\_vector(3 downto 0):= "0010";

constant ALUAND: std\_logic\_vector(3 downto 0):= "0011";

constant ALUNOT: std\_logic\_vector(3 downto 0):= "0100";

constant ALUSUB: std\_logic\_vector(3 downto 0):= "0101";

constant ALUSHR: std\_logic\_vector(3 downto 0):= "0110";

constant ALUSHL: std\_logic\_vector(3 downto 0):= "0111";

constant ALUROR: std\_logic\_vector(3 downto 0):= "1000";

constant ALUROL: std\_logic\_vector(3 downto 0):= "1001";

constant ALUY: std\_logic\_vector(3 downto 0):= "1010";

-- intruction number constants to be used with instruction\_flags slices

constant iNOP: std\_logic\_vector(7 downto 0):= "00000000";

constant iSTA: std\_logic\_vector(7 downto 0):= "00010000";

constant iLDA: std\_logic\_vector(7 downto 0):= "00100000";

constant iADD: std\_logic\_vector(7 downto 0):= "00110000";

constant iOR: std\_logic\_vector(7 downto 0):= "01000000";

constant iAND: std\_logic\_vector(7 downto 0):= "01010000";

constant iNOT: std\_logic\_vector(7 downto 0):= "01100000";

constant iSUB: std\_logic\_vector(7 downto 0):= "01110000";

constant iJMP: std\_logic\_vector(7 downto 0):= "10000000";

constant iJN: std\_logic\_vector(7 downto 0):= "10010000";

constant iJP: std\_logic\_vector(7 downto 0):= "10010100";

constant iJV: std\_logic\_vector(7 downto 0):= "10011000";

constant iJNV: std\_logic\_vector(7 downto 0):= "10011100";

constant iJZ: std\_logic\_vector(7 downto 0):= "10100000";

constant iJNZ: std\_logic\_vector(7 downto 0):= "10100100";

constant iJC: std\_logic\_vector(7 downto 0):= "10110000";

constant iJNC: std\_logic\_vector(7 downto 0):= "10110100";

constant iJB: std\_logic\_vector(7 downto 0):= "10111000";

constant iJNB: std\_logic\_vector(7 downto 0):= "10111100";

constant iSHR: std\_logic\_vector(7 downto 0):= "11100000";

constant iSHL: std\_logic\_vector(7 downto 0):= "11100001";

constant iROR: std\_logic\_vector(7 downto 0):= "11100010";

constant iROL: std\_logic\_vector(7 downto 0):= "11100011";

constant iHLT: std\_logic\_vector(7 downto 0):= "11110000";

-- BRAM memory component (mem\_ahmes)

component mem\_ahmes

port(

clka: in std\_logic;

wea: in std\_logic\_vector(0 downto 0);

addra: in std\_logic\_vector(7 downto 0);

dina: in std\_logic\_vector(7 downto 0);

douta: out std\_logic\_vector(7 downto 0));

end component;

begin -- datapath behavioral start

-- hw connections (not clock dependent)

mem\_out <= omem\_out;

mem: mem\_ahmes

port map(

clka => CLOCK,

wea => mem\_write,

addra => reg\_MA,

dina => reg\_MD,

douta => omem\_out);

PC: process(CLOCK, RESET) -- program counter register

-- standard register, has functions to reset, load values and increment internal value by 1

-- mainly recieves values from reg\_MD

-- asynchronous reset

begin

if(RESET = '1') then

reg\_PC <= "00000000";

elsif(rising\_edge(CLOCK)) then

if(load\_PC = '1') then

reg\_PC <= reg\_MD;

elsif(inc\_PC = '1') then

reg\_PC <= std\_logic\_vector(unsigned(reg\_PC) + 1);

end if;

end if;

end process;

AC: process(CLOCK, RESET, load\_AC) -- accumulator register

-- standard register, has functions to reset and load values

-- mainly recieves values from reg\_ALU

-- asynchronous reset

begin

if(RESET = '1') then

reg\_AC <= "00000000";

elsif(rising\_edge(CLOCK)) then

ALU\_X <= reg\_AC; -- retrieves ALU X from reg\_AC (accumulator)

if(load\_AC = '1') then

reg\_AC <= std\_logic\_vector(ALU\_out(7 downto 0));

end if;

end if;

end process;

-- ALU flag registers

regN: process(CLOCK, RESET, load\_N) -- negative flag register

-- standard register, has functions to reset and load values

-- mainly recieves values from internal signal flag\_N

-- asynchronous reset

begin

if(RESET = '1') then

reg\_N <= '0';

elsif(rising\_edge(CLOCK) and load\_N = '1') then

reg\_N <= flag\_N;

end if;

end process;

regZ: process(CLOCK, RESET, load\_Z) -- zero flag register

-- standard register, has functions to reset and load values

-- mainly recieves values from internal signal flag\_Z

-- asynchronous reset

begin

if(RESET = '1') then

reg\_Z <= '0';

elsif(rising\_edge(CLOCK) and load\_Z = '1') then

reg\_Z <= flag\_Z;

end if;

end process;

regV: process(CLOCK, RESET, load\_V) -- overflow flag register

-- standard register, has functions to reset and load values

-- mainly recieves values from internal signal flag\_V

-- asynchronous reset

begin

if(RESET = '1') then

reg\_V <= '0';

elsif(rising\_edge(CLOCK) and load\_V = '1') then

reg\_V <= flag\_V;

end if;

end process;

regC: process(CLOCK, RESET, load\_C) -- carry flag register

-- standard register, has functions to reset and load values

-- mainly recieves values from internal signal flag\_C

-- asynchronous reset

begin

if(RESET = '1') then

reg\_C <= '0';

elsif(rising\_edge(CLOCK) and load\_C = '1') then

reg\_C <= flag\_C;

end if;

end process;

regB: process(CLOCK, RESET, load\_B) -- borrow flag register

-- standard register, has functions to reset and load values

-- mainly recieves values from internal signal flag\_B

-- asynchronous reset

begin

if(RESET = '1') then

reg\_B <= '0';

elsif(rising\_edge(CLOCK) and load\_B = '1') then

reg\_B <= flag\_B;

end if;

end process;

IR: process (CLOCK, RESET, load\_I) -- instruction register

-- standard register, has functions to reset and load values

-- recieves values from reg\_MD

-- asynchronous reset

begin

if(RESET = '1') then

reg\_I <= "00000000";

elsif(rising\_edge(CLOCK) and load\_I = '1') then

reg\_I <= reg\_MD;

end if;

end process;

MAR: process(CLOCK, RESET, load\_MA) -- memory address register

-- standard register, has functions to reset and load values

-- recieves values from MA\_MUX\_out

-- asynchronous reset

begin

if (RESET = '1') then

reg\_MA <= "00000000";

elsif (rising\_edge(CLOCK) and load\_MA = '1') then

reg\_MA <= MAR\_MUX\_out;

end if;

end process;

MDR: process(CLOCK, RESET, load\_MD) -- memory data register

-- standard register, has functions to reset and load values

-- recieves values from MD\_MUX\_out

-- asynchronous reset

begin

if(RESET = '1') then

reg\_MD <= "00000000";

elsif(rising\_edge(CLOCK)) then

ALU\_Y <= reg\_MD; -- retrieves ALU Y from reg\_MD (memory data)

if(load\_MD = '1') then

reg\_MD <= MDR\_MUX\_out;

end if;

end if;

end process;

MAR\_MUX: process(sel\_MUX\_MAR, reg\_PC, reg\_MD) -- MAR 2x1 multiplexer

-- 2x1 MUX, selecting between reg\_PC and reg\_MD

begin

if(sel\_MUX\_MAR = "0") then

MAR\_MUX\_out <= reg\_PC;

else

MAR\_MUX\_out <= reg\_MD;

end if;

end process;

MDR\_MUX: process(sel\_MUX\_MDR, oMEM\_out, reg\_AC, mem\_read) -- MDR 2x1 multiplexer

-- 2x1 MUX, selecting between MEM\_out and reg\_AC

begin

-- requires 'mem\_read' signal to enable memory reading

if(sel\_MUX\_MDR = "0" and mem\_read = "1") then

MDR\_MUX\_out <= oMEM\_out;

else

MDR\_MUX\_out <= reg\_AC;

end if;

end process;

ALU: process(sel\_ALU, ALU\_X, ALU\_Y, flag\_C, ALU\_op) -- arithmetic logic unit

-- Performs arithmetic and logical operations on two input operands

-- Output result is stored in ALU\_op

-- Operation selected based on sel\_ALU input

-- Available operations:

-- ADD - adds ALU\_X and ALU\_Y

-- OR - logical OR of ALU\_X and ALU\_Y

-- AND - logical AND of ALU\_X and ALU\_Y

-- NOT - bitwise complement of ALU\_X

-- SUB - subtracts ALU\_Y from ALU\_X

-- SHR - right shift of ALU\_X by 1 bit

-- SHL - left shift of ALU\_X by 1 bit

-- ROR - right rotate of ALU\_X by 1 bit

-- ROL - left rotate of ALU\_X by 1 bit

-- NOP - no operation, ALU\_op receives ALU\_Y

begin

if(unsigned(ALU\_op(7 downto 0)) = 0) then

flag\_Z <= '1';

else

flag\_Z <= '0';

end if;

ALU\_out <= std\_logic\_vector(ALU\_op(7 downto 0)); -- updates ALU output with 7 lsb of ALU\_op

flag\_V <= '1';

flag\_B <= '1';

flag\_N <= ALU\_op(7); -- accumulator msb

flag\_C <= ALU\_op(8); -- carry flag

case sel\_ALU is

when ALUADD => -- ADD

ALU\_op <= std\_logic\_vector(unsigned('0' & ALU\_X) + unsigned('0' & ALU\_Y));

-- checks for overflow

if (ALU\_X(7) = '0' and ALU\_Y(7) = '0' and ALU\_op(7) = '1') then

flag\_V <= '1';

else

flag\_V <= '0';

end if;

-- resets other flag signals

flag\_B <= '0';

when ALUOR => -- OR

ALU\_op <= std\_logic\_vector(('0' & ALU\_X) or ('0' & ALU\_Y));

-- resets other flag signals

flag\_B <= '0';

flag\_V <= '0';

when ALUAND => -- AND

ALU\_op <= std\_logic\_vector(('0' & ALU\_X) and ('0' & ALU\_Y));

-- resets other flag signals

flag\_B <= '0';

flag\_V <= '0';

when ALUNOT => -- NOT

ALU\_op <= std\_logic\_vector(not('0' & ALU\_X));

-- resets other flag signals

flag\_B <= '0';

flag\_V <= '0';

when ALUSUB => -- SUB

ALU\_op <= std\_logic\_vector(unsigned('0' & ALU\_X) - unsigned('0' & ALU\_Y));

-- checks for overflow

if (ALU\_X(7) = '1' and ALU\_Y(7) = '1' and ALU\_op(7) = '0') then

flag\_V <= '1';

else

flag\_V <= '0';

end if;

-- borrow flag

flag\_B <= ALU\_op(7) and ALU\_op(8);

when ALUSHR => -- SHR

ALU\_op(8) <= ALU\_X(0); -- ALU\_op, and by consequence carry recieves reg\_AC lsb

ALU\_op(7) <= '0'; -- ULA\_out msb recieves 0

ALU\_op(6) <= ALU\_X(7);

ALU\_op(5) <= ALU\_X(6);

ALU\_op(4) <= ALU\_X(5);

ALU\_op(3) <= ALU\_X(4);

ALU\_op(2) <= ALU\_X(3);

ALU\_op(1) <= ALU\_X(2);

ALU\_op(0) <= ALU\_X(1);

-- resets other flag signals

flag\_B <= '0';

flag\_V <= '0';

when ALUSHL => -- SHL

ALU\_op(8) <= ALU\_X(7); -- ALU\_op, and by consequence carry recieves reg\_AC msb

ALU\_op(7) <= ALU\_X(6);

ALU\_op(6) <= ALU\_X(5);

ALU\_op(5) <= ALU\_X(4);

ALU\_op(4) <= ALU\_X(3);

ALU\_op(3) <= ALU\_X(2);

ALU\_op(2) <= ALU\_X(1);

ALU\_op(1) <= ALU\_X(0);

ALU\_op(0) <= '0'; -- ULA\_out lsb recieves 0

-- resets other flag signals

flag\_B <= '0';

flag\_V <= '0';

when ALUROR => -- ROR

ALU\_op(8) <= ALU\_X(0);

ALU\_op(7) <= flag\_C;

ALU\_op(6) <= ALU\_X(7);

ALU\_op(5) <= ALU\_X(6);

ALU\_op(4) <= ALU\_X(5);

ALU\_op(3) <= ALU\_X(4);

ALU\_op(2) <= ALU\_X(3);

ALU\_op(1) <= ALU\_X(2);

ALU\_op(0) <= ALU\_X(1);

-- resets other flag signals

flag\_B <= '0';

flag\_V <= '0';

when ALUROL => -- ROL

ALU\_op(8) <= ALU\_X(7);

ALU\_op(7) <= ALU\_X(6);

ALU\_op(6) <= ALU\_X(5);

ALU\_op(5) <= ALU\_X(4);

ALU\_op(4) <= ALU\_X(3);

ALU\_op(3) <= ALU\_X(2);

ALU\_op(2) <= ALU\_X(1);

ALU\_op(1) <= ALU\_X(0);

ALU\_op(0) <= flag\_C;

-- resets other flag signals

flag\_B <= '0';

flag\_V <= '0';

when ALUY => -- ULAY

ALU\_op <= std\_logic\_vector('0' & ALU\_Y);

-- resets other flag signals

flag\_B <= '0';

flag\_V <= '0';

when others => -- NOP (ULAX - reg\_AC)

ALU\_op <= '0' & ALU\_X;

end case;

end process;

IR\_DECOD: process(reg\_I, IR\_DECOD\_out) -- IR\_DECOD, used for setting instruction\_flags

-- decodes reg\_I signal setting a 24 bits vector, with flags for each of

-- the processor instructions

begin

instruction\_flags <= IR\_DECOD\_out;

IR\_DECOD\_out <= "000000000000000000000000";

case reg\_I is

when iNOP => IR\_DECOD\_out <= "000000000000000000000001"; -- 00 NOP

when iSTA => IR\_DECOD\_out <= "000000000000000000000010"; -- 16 STA

when iLDA => IR\_DECOD\_out <= "000000000000000000000100"; -- 32 LDA

when iADD => IR\_DECOD\_out <= "000000000000000000001000"; -- 48 ADD

when iOR => IR\_DECOD\_out <= "000000000000000000010000"; -- 64 OR

when iAND => IR\_DECOD\_out <= "000000000000000000100000"; -- 80 AND

when iNOT => IR\_DECOD\_out <= "000000000000000001000000"; -- 96 NOT

when iSUB => IR\_DECOD\_out <= "000000000000000010000000"; -- 112 SUB

when iJMP => IR\_DECOD\_out <= "000000000000000100000000"; -- 128 JMP

when iJN => IR\_DECOD\_out <= "000000000000001000000000"; -- 144 JN

when iJP => IR\_DECOD\_out <= "000000000000010000000000"; -- 148 JP

when iJV => IR\_DECOD\_out <= "000000000000100000000000"; -- 152 JV

when iJNV => IR\_DECOD\_out <= "000000000001000000000000"; -- 156 JNV

when iJZ => IR\_DECOD\_out <= "000000000010000000000000"; -- 160 JZ

when iJNZ => IR\_DECOD\_out <= "000000000100000000000000"; -- 164 JNZ

when iJC => IR\_DECOD\_out <= "000000001000000000000000"; -- 176 JC

when iJNC => IR\_DECOD\_out <= "000000010000000000000000"; -- 180 JNC

when iJB => IR\_DECOD\_out <= "000000100000000000000000"; -- 184 JB

when iJNB => IR\_DECOD\_out <= "000001000000000000000000"; -- 188 JNB

when iSHR => IR\_DECOD\_out <= "000010000000000000000000"; -- 224 SHR

when iSHL => IR\_DECOD\_out <= "000100000000000000000000"; -- 225 SHL

when iROR => IR\_DECOD\_out <= "001000000000000000000000"; -- 226 ROR

when iROL => IR\_DECOD\_out <= "010000000000000000000000"; -- 227 ROL

when iHLT => IR\_DECOD\_out <= "100000000000000000000000"; -- 240 HLT

when others => IR\_DECOD\_out <= "100000000000000000000000"; -- 00 HLT

end case;

end process;

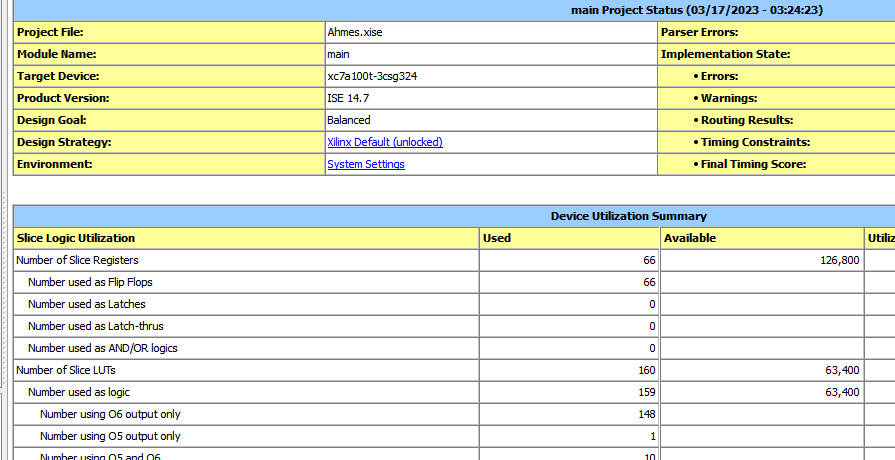
end Behavioral;

**Qual componente FPGA escolheste para a síntese? Xc7a100t-3csg324**

**Quantos registradores tem o datapath do RAMSES? 10**

**Quantas operações diferentes tem a ULA? 11**

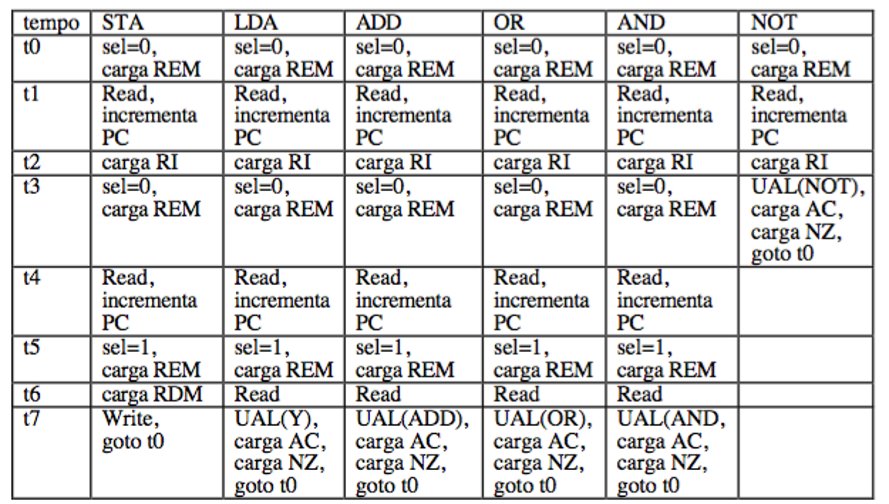
**A área do DATAPTH em # LUTs: 160 e #ffps: 66**

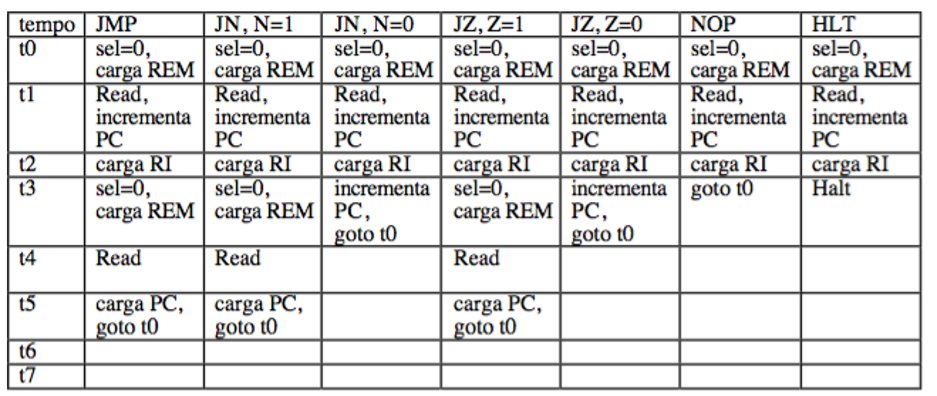


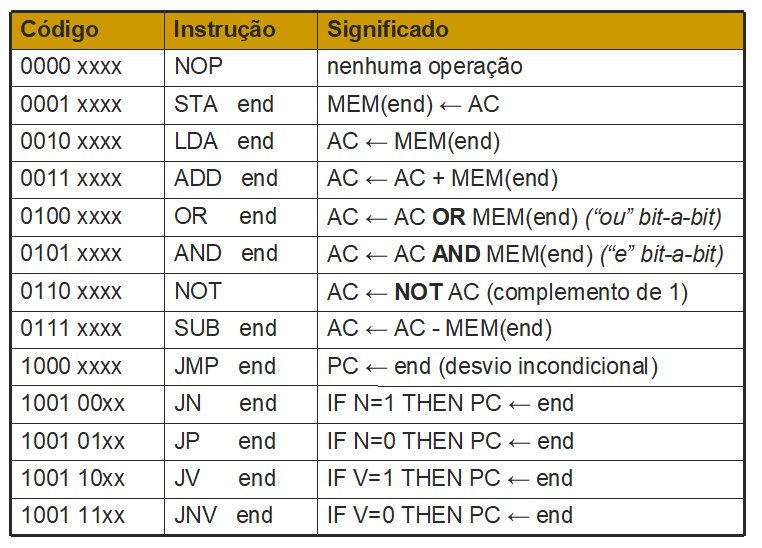
**PASSO 2:** 3 pontos

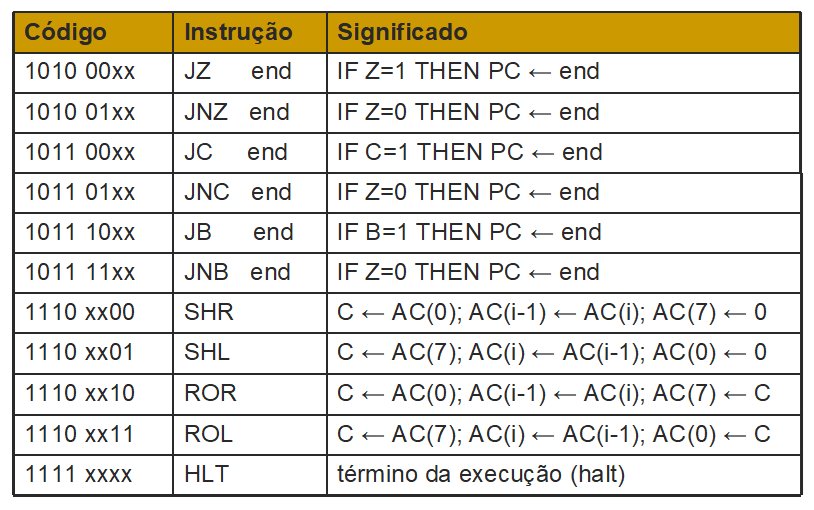
Descrever a parte de controle do AHMES em VHDL como uma maquina de estados.

Dada as tabelas com as instruções do Neander por estado da máquina de estrados









Completar a tabela a seguir com as instruções AHMES que não tem no NEANDER

| Tempo | SHR | SHL | ROR | ROL | SUB |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| T0 |  |  |  |  |  |  |  |
| T1 |  |  |  |  |  |  |  |
| T2 |  |  |  |  |  |  |  |
| T3 |  |  |  |  |  |  |  |
| T4 |  |  |  |  |  |  |  |
| T5 |  |  |  |  |  |  |  |
| T6 |  |  |  |  |  |  |  |
| T7 |  |  |  |  |  |  |  |

E as instruções novas de Desvio

| Tempo |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| T0 |  |  |  |  |  |  |  |
| T1 |  |  |  |  |  |  |  |
| T2 |  |  |  |  |  |  |  |
| T3 |  |  |  |  |  |  |  |
| T4 |  |  |  |  |  |  |  |
| T5 |  |  |  |  |  |  |  |
| T6 |  |  |  |  |  |  |  |
| T7 |  |  |  |  |  |  |  |

Cole aqui o VHDL da parte de controle usando FSM com dois process.

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity control is

Port(

CLOCK: in std\_logic;

RESET: in std\_logic;

instruction\_flags: in std\_logic\_vector(23 downto 0);

-- ALU flags

reg\_N: in std\_logic;

reg\_Z: in std\_logic;

reg\_V: in std\_logic;

reg\_C: in std\_logic;

reg\_B: in std\_logic;

-- register control outputs

inc\_PC: out std\_logic;

load\_AC: out std\_logic;

load\_PC: out std\_logic;

load\_MA: out std\_logic;

load\_MD: out std\_logic;

load\_I: out std\_logic;

load\_N: out std\_logic;

load\_Z: out std\_logic;

load\_V: out std\_logic;

load\_C: out std\_logic;

load\_B: out std\_logic;

-- selectors

sel\_MUX\_MAR: out std\_logic\_vector(0 downto 0);

sel\_MUX\_MDR: out std\_logic\_vector(0 downto 0);

sel\_ALU: out std\_logic\_vector(3 downto 0);

-- memory control

mem\_read: out std\_logic\_vector(0 downto 0);

mem\_write: out std\_logic\_vector(0 downto 0));

end control;

architecture Behavioral of control is

type state\_type is (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11);

signal next\_state, current\_state: state\_type;

-- intruction signals to be used with instruction\_flags slices

signal oNOP: std\_logic;

signal oSTA: std\_logic;

signal oLDA: std\_logic;

signal oADD: std\_logic;

signal oOR: std\_logic;

signal oAND: std\_logic;

signal oNOT: std\_logic;

signal oSUB: std\_logic;

signal oJN: std\_logic;

signal oJP: std\_logic;

signal oJV: std\_logic;

signal oJNV: std\_logic;

signal oJZ: std\_logic;

signal oJNZ: std\_logic;

signal oJC: std\_logic;

signal oJNC: std\_logic;

signal oJB: std\_logic;

signal oJNB: std\_logic;

signal oSHR: std\_logic;

signal oSHL: std\_logic;

signal oROR: std\_logic;

signal oROL: std\_logic;

signal oHLT: std\_logic;

-- ALU operation constants

constant ALUNOP: std\_logic\_vector(3 downto 0):= "0000";

constant ALUADD: std\_logic\_vector(3 downto 0):= "0001";

constant ALUOR: std\_logic\_vector(3 downto 0):= "0010";

constant ALUAND: std\_logic\_vector(3 downto 0):= "0011";

constant ALUNOT: std\_logic\_vector(3 downto 0):= "0100";

constant ALUSUB: std\_logic\_vector(3 downto 0):= "0101";

constant ALUSHR: std\_logic\_vector(3 downto 0):= "0110";

constant ALUSHL: std\_logic\_vector(3 downto 0):= "0111";

constant ALUROR: std\_logic\_vector(3 downto 0):= "1000";

constant ALUROL: std\_logic\_vector(3 downto 0):= "1001";

constant ALUY: std\_logic\_vector(3 downto 0):= "1010";

begin

-- hw connections (slices instruction\_flags to internal instruction signals)

oNOP <= instruction\_flags(0);

oSTA <= instruction\_flags(1);

oLDA <= instruction\_flags(2);

oADD <= instruction\_flags(3);

oOR <= instruction\_flags(4);

oAND <= instruction\_flags(5);

oNOT <= instruction\_flags(6);

oSUB <= instruction\_flags(7);

oJN <= instruction\_flags(9);

oJP <= instruction\_flags(10);

oJV <= instruction\_flags(11);

oJNV <= instruction\_flags(12);

oJZ <= instruction\_flags(13);

oJNZ <= instruction\_flags(14);

oJC <= instruction\_flags(15);

oJNC <= instruction\_flags(16);

oJB <= instruction\_flags(17);

oJNB <= instruction\_flags(18);

oSHR <= instruction\_flags(19);

oSHL <= instruction\_flags(20);

oROR <= instruction\_flags(21);

oROL <= instruction\_flags(22);

oHLT <= instruction\_flags(23);

SC: process(CLOCK, RESET) -- FSM state control

begin

if(RESET = '1') then

current\_state <= S0;

elsif(rising\_edge(CLOCK)) then

current\_state <= next\_state;

end if;

end process;

FSM: process(

current\_state,

-- value flags

reg\_N,

reg\_Z,

reg\_V,

reg\_C,

reg\_B,

-- instruction flags

oNOP,

oSTA,

oLDA,

oADD,

oOR,

oAND,

oNOT,

oSUB,

oJN,

oJP,

oJV,

oJNV,

oJZ,

oJNZ,

oJC,

oJNC,

oJB,

oJNB,

oSHR,

oSHL,

oROR,

oROL,

oHLT)

-- FSM description

begin

-- resets signals

inc\_PC <= '0';

load\_AC <= '0';

load\_PC <= '0';

load\_MA <= '0';

load\_MD <= '0';

load\_I <= '0';

load\_N <= '0';

load\_Z <= '0';

load\_V <= '0';

load\_C <= '0';

load\_B <= '0';

sel\_MUX\_MAR <= "0";

sel\_MUX\_MDR <= "0";

mem\_write <= "0";

mem\_read <= "0";

sel\_ALU <= ALUNOP; -- default ALU operation

case current\_state is

when S0 => -- POINTS

-- updates memory cursor

sel\_MUX\_MAR <= "0"; -- reg\_MA <= reg\_PC

load\_MA <= '1'; -- updates MAR

-- goto S1

next\_state <= S1;

when S1 => -- SKIP

-- goto S2

next\_state <= S2;

when S2 => -- READS

-- reads byte from memory updating MD

mem\_read <= "1";

sel\_MUX\_MDR <= "0"; -- reg\_MD <= mem\_out

load\_MD <= '1';

-- increments reg\_PC

inc\_PC <= '1'; -- adjusts memory cursor and program counter

-- goto S3

next\_state <= S3;

when S3 => -- FETCH

-- fetches new instruction

load\_I <= '1'; -- reg\_I <= reg\_MD

-- goto s3

next\_state <= S4;

when S4 => -- INS/POINT

if(oNOP = '1') then

-- goto S0

next\_state <= S0;

elsif(oNOT = '1') then

-- updates ALU operation

sel\_ALU <= ALUNOT;

-- updates AC value

load\_AC <= '1';

-- updates flag signals

load\_N <= '1';

load\_Z <= '1';

load\_V <= '1';

load\_C <= '1';

load\_B <= '1';

-- goto S0

next\_state <= S0;

elsif(oJN = '1' and reg\_N = '0') then -- JN if n=0

-- conditions does not match for jumping to given addres

-- goes to the next instruction, going to S0

inc\_PC <= '1';

next\_state <= S0;

elsif(oJP = '1' and reg\_N = '1') then -- JP if n=1

-- conditions does not match for jumping to given addres

-- goes to the next instruction, going to S0

inc\_PC <= '1';

next\_state <= S0;

elsif(oJV = '1' and reg\_V = '0') then -- JV if v=0

-- conditions does not match for jumping to given addres

-- goes to the next instruction, going to S0

inc\_PC <= '1';

next\_state <= S0;

elsif(oJNV = '1' and reg\_V = '1') then -- JNV if v=1

-- conditions does not match for jumping to given addres

-- goes to the next instruction, going to S0

inc\_PC <= '1';

next\_state <= S0;

elsif(oJZ = '1' and reg\_Z = '0') then -- JZ if z=0

-- conditions does not match for jumping to given addres

-- goes to the next instruction, going to S0

inc\_PC <= '1';

next\_state <= S0;

elsif(oJNZ = '1' and reg\_Z = '1') then -- JNZ if z=1

-- conditions does not match for jumping to given addres

-- goes to the next instruction, going to S0

inc\_PC <= '1';

next\_state <= S0;

elsif(oJC = '1' and reg\_C = '0') then -- JC if c=0

-- conditions does not match for jumping to given addres

-- goes to the next instruction, going to S0

inc\_PC <= '1';

next\_state <= S0;

elsif(oJNC = '1' and reg\_C = '1') then -- JNC if c=1

-- conditions does not match for jumping to given addres

-- goes to the next instruction, going to S0

inc\_PC <= '1';

next\_state <= S0;

elsif(oJB = '1' and reg\_B = '0') then -- JB if b=0

-- conditions does not match for jumping to given addres

-- goes to the next instruction, going to S0

inc\_PC <= '1';

next\_state <= S0;

elsif(oJNB = '1' and reg\_B = '1') then -- JNB if b=1

-- conditions does not match for jumping to given addres

-- goes to the next instruction, going to S0

inc\_PC <= '1';

next\_state <= S0;

elsif(oSHR = '1') then -- SHR

-- updates ALU instruction

sel\_ALU <= ALUSHR;

-- updates accumulator value

load\_AC <= '1';

-- updates logic flag signals

load\_N <= '1';

load\_Z <= '1';

load\_V <= '1';

load\_C <= '1';

load\_B <= '1';

-- goes to S0

next\_state <= S0;

elsif(oSHL = '1') then -- SHL

-- updates ALU instruction

sel\_ALU <= ALUSHL;

-- updates accumulator value

load\_AC <= '1';

-- updates logic flag signals

load\_N <= '1';

load\_Z <= '1';

load\_V <= '1';

load\_C <= '1';

load\_B <= '1';

-- goes to S0

next\_state <= S0;

elsif(oROR = '1') then -- ROR

-- updates ALU instruction

sel\_ALU <= ALUROR;

-- updates accumulator value

load\_AC <= '1';

-- updates logic flag signals

load\_N <= '1';

load\_Z <= '1';

load\_V <= '1';

load\_C <= '1';

load\_B <= '1';

-- goes to S0

next\_state <= S0;

elsif(oROL = '1') then -- ROL

-- updates ALU instruction

sel\_ALU <= ALUROL;

-- updates accumulator value

load\_AC <= '1';

-- updates logic flag signals

load\_N <= '1';

load\_Z <= '1';

load\_V <= '1';

load\_C <= '1';

load\_B <= '1';

-- goes to S0

next\_state <= S0;

elsif(oHLT = '1') then -- HLT

-- goto S8 (loop state)

next\_state <= S8;

else

-- for any sucessful jumping instruction and others

-- sets memory cursor position to current reg\_PC value

sel\_MUX\_MAR <= "0"; -- reg\_MA <= reg\_PC

load\_MA <= '1'; -- updates reg\_MA with reg\_PC

sel\_MUX\_MDR <= "1";

next\_state <= S5;

end if;

when S5 => -- SKIP

-- goto S6

next\_state <= S6;

when S6 => -- SKIP

-- reads a new byte from memory

mem\_read <= "1";

sel\_MUX\_MDR <= "0"; -- reg\_MD <= mem\_out

load\_MD <= '1'; -- updates memory data register

-- increments reg\_PC for any of the following instructions

if(oSTA = '1' or oLDA = '1' or oADD = '1' or oOR = '1' or oAND = '1' or oSUB = '1') then

inc\_PC <= '1';

else

inc\_PC <= '0';

end if;

-- goes to S7

next\_state <= S7;

when S7 => -- POINT

if(oSTA = '1' or oLDA = '1' or oADD = '1' or oOR = '1' or oAND = '1' or oSUB = '1') then

sel\_MUX\_MAR <= "1"; -- reg\_MA <= reg\_MD

load\_MA <= '1'; -- updates reg\_MA value

-- goto S8

next\_state <= S8;

else -- for any other instruction (jumps)

-- updates program counter

load\_PC <= '1';

-- goto S0

next\_state <= S0;

end if;

when S8 => -- SKIP

next\_state <= S9;

when S9 => -- READ

if(oSTA = '1') then -- STA

sel\_MUX\_MDR <= "1"; -- reg\_MD <= reg\_AC

load\_MD <= '1'; -- updates reg\_MD value

else

-- for any other instruction

mem\_read <= "1";

sel\_MUX\_MDR <= "0"; -- reg\_MD <= mem\_out

load\_MD <= '1'; -- updates MD value

end if;

-- goto s10

next\_state <= S10;

when S10 => -- INS

if(oSTA = '1') then

mem\_write <= "1";

-- goto S0

next\_state <= S0;

elsif(oLDA = '1') then

-- updates ULA\_out and by consequence the accumulator with reg\_MD

load\_AC <= '1';

-- updates value flag registers

load\_N <= '1';

load\_Z <= '1';

load\_V <= '1';

load\_C <= '1';

load\_B <= '1';

-- updates ALU instruction

sel\_ALU <= ALUY;

-- goto S0

next\_state <= S0;

elsif(oADD = '1') then

-- updates ALU instruction

sel\_ALU <= ALUADD;

-- updates the accumulator with ULA\_out

load\_AC <= '1';

-- updates value flag registers

load\_N <= '1';

load\_Z <= '1';

load\_V <= '1';

load\_C <= '1';

load\_B <= '1';

-- goto S0

next\_state <= S0;

elsif(oOR = '1') then

-- updates ALU instruction

sel\_ALU <= ALUOR;

-- updates the accumulator with ULA\_out

load\_AC <= '1';

-- updates value flag registers

load\_N <= '1';

load\_Z <= '1';

load\_V <= '1';

load\_C <= '1';

load\_B <= '1';

-- goto S0

next\_state <= S0;

elsif(oAND = '1') then

-- updates ALU instruction

sel\_ALU <= ALUAND;

-- updates the accumulator with ULA\_out

load\_AC <= '1';

-- updates value flag registers

load\_N <= '1';

load\_Z <= '1';

load\_V <= '1';

load\_C <= '1';

load\_B <= '1';

-- goto S0

next\_state <= S0;

elsif(oSUB = '1') then

-- updates ALU instruction

sel\_ALU <= ALUSUB;

-- updates the accumulator with ULA\_out

load\_AC <= '1';

-- updates value flag registers

load\_N <= '1';

load\_Z <= '1';

load\_V <= '1';

load\_C <= '1';

load\_B <= '1';

-- goto S0

next\_state <= S0;

else

-- invalid instruction

next\_state <= S11;

end if;

when S11 => -- HALT

-- loop state (HALT)

next\_state <= S10;

when others => -- HALT

next\_state <= S10;

end case;

end process;

end Behavioral;

**PASSO 3:** 1 ponto

Descrever o programa em Assembly do AHMES que realize a multiplicação de dois números inteiros positivos de 8 bits por Deslocamento e soma em binário e colocar no arquivo .COE na memória BRAM.

Inserir aqui o programa em Assembly com explicação

; programa multiplicação 10x2

LDA 128

SHL

HLT

ORG 128

Cte\_10:

DB 10

Por deslocamento:  
memory\_initialization\_radix=10;

memory\_initialization\_vector=0,32,128,225,240,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;

; programa multiplicação 10x2

LDA 128

ADD cte\_10

HLT

ORG 128

Cte\_10:

DB 10

Por somas sucessivas:

memory\_initialization\_radix=10;

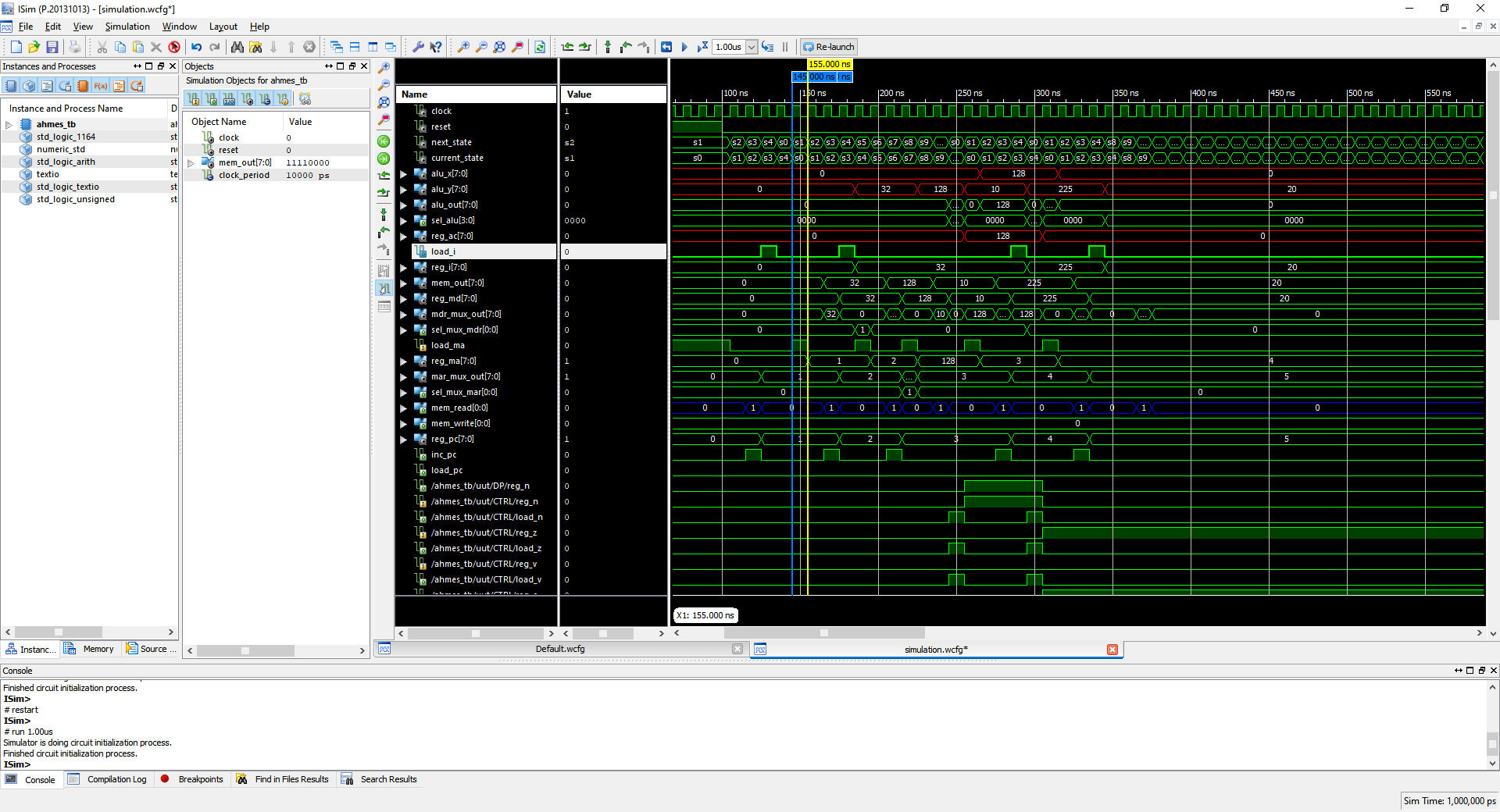
memory\_initialization\_vector=0,32,128,48,128,240,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;

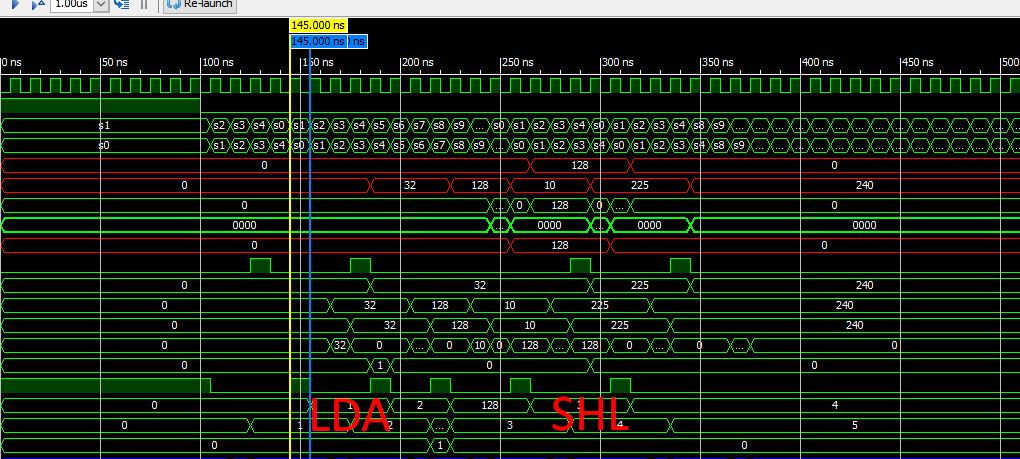
**PASSO 4:** 3 pontos

Simular sem atraso o AHMES com um programa teste a ser feito pelo aluno e depois que testado e funcionando, simular com o programa do passo 3. Depois de tudo funcionando, simular também com atraso.

Lembrem-se que deve ser feito um testbench para a simulação.

Colar aqui o programa teste e simulações (.JPG)





**Quantos ciclos de relógio foram necessários para a execução do programa de multiplicação no AHMES?** 38 c.c (programa anterior – até atingir o HLT)